MATLAB Companion Script for *Machine Learning* ex6 (Optional)

Introduction

Coursera's *Machine Learning* was designed to provide you with a greater understanding of machine learning algorithms- what they are, how they work, and where to apply them. You are also shown techniques to improve their performance and to address common issues. As is mentioned in the course, there are many tools available that allow you to use machine learning algorithms *without* having to implement them yourself. This Live Script was created by MathWorks to help *Machine Learning* students explore the data analysis and machine learning tools available in MATLAB.

FAQ

Who is this intended for?

- This script is intended for students using MATLAB Online who have completed ex6 and want to learn more about the corresponding machine learning tools in MATLAB.

How do I use this script?

- In the sections that follow, read the information provided about the data analysis and machine learning tools in MATLAB, then run the code in each section and examine the results. You may also be presented with instructions for using a MATLAB machine learning app. This script should be located in the ex6 folder which should be set as your Current Folder in MATLAB Online.

Can I use the tools in this companion script to complete the programming exercises?

- No. Most algorithm steps implemented in the programming exercises are handled automatically by MATLAB machine learning functions. Additionally, the results will be similar, but not identical, to those in the programming exercises due to differences in implementation, parameter settings, and randomization.

Where can I obtain help with this script or report issues?

- As this script is not part of the original course materials, please direct any questions, comments, or issues to the *MATLAB Help* discussion forum.

Support Vector Machines

In this Live Script, we will create and train support vector machine classification models using apps and functions from the Statistics and Machine Learning Toolbox.

Files needed for this script:

- `ex6data1.mat` - Example Dataset 1
- `ex6data2.mat` - Example Dataset 2
- `ex6data3.mat` - Example Dataset 3
- `spamTrain.mat` - Spam training set

- `spamTest.mat` - Spam test set
- `emailSample1.txt` - Sample email 1
- `emailSample2.txt` - Sample email 2
- `spamSample1.txt` - Sample spam 1
- `spamSample2.txt` - Sample spam 2
- `vocab.txt` - Vocabulary list

Table of Contents

Train a Linear SVM with the Classification Learner App

In this section we train and export an linear SVM classifier model using the Classification Learner App.

Load dataset 1

Run the code below to load example dataset 1, which consists of two feature variables in the matrix X and the binary response vector y. The variables are concatenated into a single data matrix, `data` for use with the Classification Learner App.

```
clear;
load ex6data1.mat;
data = [X,y];
```

Follow the steps in the next few sections to train and export an SVM classifier.

Note: If you have difficulty reading the instructions below while the app is open in MATLAB Online, export this script to a pdf file which you can then use to display the instructions in a seperate browser tab or window. To export this script, click on the 'Save' button in the 'Live Editor' tab above, then select 'Export to PDF'.

Open the app and select the predictor and response variables

1. In the MATLAB Apps tab, select the '**Classification Learner'** from the '**Machine Learning'** section (you may need to expand the menu of available apps).
2. Select '**New Session -> From Workspace**' to start a new interactive session.
3. Under '**Workspace Variable'**, select '**data**'.
4. Under '**Validation**', select '**Cross-Validation**' and use the slider to select 10 folds (see the companion script for ex5 for more information on K-fold cross-validation)
5. Click the '**Start Session**' button.

Select and train the model

1. In the model list, the default model is 'Fine Tree'. Expand the model list and select '**Linear SVM**' from the '**Support Vector Machines**' list.
2. Select '**Train**' to train the model. Note that the data is automatically normalized before training.

Evaluate the model

After training, there are several options available for evaluating the model's performance.

- The '**Scatter Plot**' is automatically updated to indicate misclassified objects with an 'x' marker.
- The '**Confusion Matrix**' plot contains the correct and incorrect model predictions by class.
- The '**Current Model**' pane contains a summary of the model as well as the training accuracy, training time, and prediction speed for comparison with other models.

Export the model to the workspace and extract the model variable

1. Select '**Export Model -> Export Model**'.
2. Select the default output variable name ('trainedModel') and click '**OK**'.

3. Run the code below to extract the classifier model into the variable `linSVMmdl`

```
linSVMmdl = trainedModel.ClassificationSVM
```

Predict classes and visualize the decision boundary using the `predict` function

As with the other model variables discussed previously in the companion scripts, all of the information about the SVM model is included in the model variable- in this case a classificationSVM model variable. This includes the training data and information, model coefficients, normalization coefficients, etc. The model variable can then be used to predict the class of new feature data using the predict function. Run the code below to predict the class of a random training example and plot the decision boundary using the decisionBoundary function (included at the end of this script as a local function) for comparison with your results from ex6. Note that normalization was automatically implemented by the Classification Learner App before training the SVM model, there is no need to normalize feature data before passing it to predict, the data will again be normalized automatically before prediction.

```
idx = randi(length(y));
fprintf('Data Point: %d | True class: %d | Predicted class: %d',idx,y(idx),predict(linSVMmdl, X
decisionBoundary(linSVMmdl,X,y);
```

Train a Gaussian Kernel SVM with the Classification Learner App

Next, we'll create, train, and export a *non-linear* classification model using a Gaussian kernel.

Load dataset 2

Run the code below to clear the previous variables and load dataset 2. The feature and response arrays are again combined into a single matrix, `data`.

```
clear;
load ex6data2.mat;
data = [X,y];
```

Train a Gaussian kernel model

To fit a Gaussian kernel SVM classifier to dataset 2, repeat the steps in the previous section ***except***:

- Select the '**Fine Gaussian**' model* instead of **Linear SVM**' from the '**Support Vector Machines**' model list.

*Alternative Gaussian kernels are available, include 'Medium' and 'Coarse', which differ in the value of `KernelScale` model parameter. This parameter is discussed later in this script and its value can be set manually in the Classification Learner App, if desired.

Extract the model and visualize the decision boundary

Exporting the trained model to the variable `trainedModel`, then run the code below to extract the classifier model into the variable `gaussSVMmdl` and plot the decision boundary.

```
gaussSVMmdl = trainedModel.ClassificationSVM
decisionBoundary(gaussSVMmdl,X,y)
```

Fit an SVM model and explore the effect of model parameters using `fitcsvm`

In ex6, you explored the effect of the SVM parameters C and σ on the model performance. In this section, we train a cross-validated SVM programmatically using the `fitcsvm` function, where the C and σ parameters effectively correspond to the `fitcsvm` options `'BoxConstraint'` and `'KernelScale'` respectively.

Load dataset 3

Run the code below to clear the current model and load dataset 3. (After completing this section, you can repeat the analysis by select one of the first two data sets.)

```
clear;
dataset = 3;
load(['ex6data' num2str(dataset) '.mat']);
```

Train an SVM classifier with custom parameter settings

Select values for the `BoxConstraint` and `KernelScale`, then run the code in this section to train an SVM classifier using `fitcsvm` to examine the effect of the parameters on the model. A warning may be displayed for certain parameter choices if the decision boundary is not rendered (when all examples are predicted to have the same class).

```
kernelscale = 1;
boxconstraint = 1;
gaussSVMmdl = fitcsvm(X,y,...
                'BoxConstraint',boxconstraint,...
                'KernelScale',kernelscale,...
                'Standardize',true,...
                'KernelFunction','gaussian');

decisionBoundary(gaussSVMmdl,X,y);
title(sprintf('Training accuracy: %0.2f%%', 100*mean(predict(gaussSVMmdl,X)==y)));
```

Note: The `BoxConstraint` and `KernelScale` parameters are also customizable in the Classification Learner App. After selecting an SVM classification model, click the '**Advanced**' button to change these parameter values. (The `KernelScale` option is not available for linear kernels).

Automatically select `BoxConstraint` and `KernelScale` using hyperparameter optimization

In the previous section you likely found several pairs of parameter values that result in good training performance and which are also likely generalize well to new data. This is due to the similar effects these parameters have on the final model. In this section we automatically select `BoxConstraint` and `KernelScale` using hyperparameter optimization. Hyperparameter optimization was discussed previously in the companion script for ex5. Run the code below to create an options structure for K-fold cross-validation and to increase the number of optimization steps to 60. The `fitcsvm` function is then called with `BoxConstraint` and `KernelScale` parameters selected for optimization. (This may take a minute to train). The final model returned by `fitcsvm` corresponds to the one with the optimal hyperparameters*.

```
folds = 5;
opts = struct('kfold',folds,'MaxObjectiveEvaluations',60,'Repartition',true);
gaussSVMmdl = fitcsvm(X,y,...
                      'KernelFunction','gaussian',...
                      'Standardize',true,...
                      'OptimizeHyperparameters',{'KernelScale','BoxConstraint'},...
                      'HyperparameterOptimizationOptions',opts)
```

***Note:** As there are two parameters being optimized the cost function is visualized *as a surface* over a grid of `BoxConstraint` and `KernelScale` values. Furthermore, the range of parameter combinations resulting in acceptable performance observed in the previous section is seen as the 'trough' in the 'Objective function model' plot. Along this trough are parameter pairs that will provide similar performance as judged by the validation cost. Due to the random nature of validation set selection, your optimal parameter pair may vary along this region upon successive training/parameter optimizations but should still result in comparable performance.

Examine the results

Run the code below to plot the decision boundary of the optimized SVM classifier.

```
decisionBoundary(gaussSVMmdl,X,y);
title(sprintf('Training accuracy: %0.2f%%', 100*mean(predict(gaussSVMmdl,X)==y)));
```

Text Processing and Spam Email Classification

In the second part of ex6, you preprocessed and stemmed text data in the form of email samples, then trained an SVM to classify the samples as spam or not spam. In this section we introduce MATLAB `string` variables and use them to preprocess text data before training a spam classifier.

MATLAB `string` variables

string arrays provide a rich set of functionality for processing and analyzing textual data. They are now the preferred text datatype in MATLAB and offer advantages in most cases over character arrays (`char`) and cell arrays of `char` variables. Use the control below to list and obtain descriptions of the functions available for working with `strings`. Note that certain methods also have equivalent operators. For example, '+' and '==' can be used in place of `plus` and `eq`.

```
help string.strip
```

Load text data

Run the code below to load a sample email into a character vector `contents`. You can return and select a different sample to see the results of the preprocessing operations in the next section.

```
clear;
file = 'emailSample1.txt';
contents = fileread(file);
```

Preprocess text using `string` methods and functions

In this section, we will convert the email contents to a string and preprocess the email using the same steps as in the ex6 function `processEmail.` If you are curious, you can remove the semicolon after one or more commands to examine the effects of the processing steps below. These steps have been collected into the local function `preprocess` at the end of this script which will be used in the remainder of this Live Script.

- Convert `contents` to a `string`

```
processed = string(contents);
```

- Lower-case the string using `lower`.

```
processed  = lower(processed);
```

- Strip HTML tags using `eraseBetween`

```
processed  = eraseBetween(processed,'<','>','Boundaries','inclusive');
```

- Replace dollar signs with 'dollar'

```
processed = replace(processed,"$","dollar");
```

- Split the email on whitespace into individual tokens (`contents` will become a `string` *array*)

```
processed = split(processed);
```

- Normalize URLs using `contains` and logical indexing

```
processed(startsWith(processed,["http://","https://"])) =  "httpaddr";
```

- Normalize email addresses and numbers, and remove punctuation using regexprep.

```
processed = regexprep(processed,'.+@.+',"emailaddr");
processed = regexprep(processed,'[0-9]+',"number");
processed = regexprep(processed,'[^a-zA-Z0-9]',"");
```

7

- Remove empty or single character words using `strlength`

```
processed(strlength(processed)<=1) = [];
```

Stem the words in the string array

Run the code below to stem the words using `porterStemmer.m` (included with ex6). Since `porterStemmer` was created for character arrays, we'll convert the elements of `contents` to character vectors before passing them to `porterStemmer`, then convert the output back into a `string`.

```
processed = arrayfun(@(str)string(porterStemmer(char(str))),processed)
```

Import the vocabulary list

The next step is to load the vocabulary words in vocab.txt. Since the file contains the (unneeded) word indices as well as the word list, we'll import the data into a `table` usng `readtable`, then select only the words by extracting the second column and converting it into a `string` array.

```
vocab = readtable('vocab.txt');
vocab = string(vocab{:,2});
```

Map the string array values to the vocabulary list

Run the code below to create a binary vector, `features` where `features(i)` equals 1 if `vocab(i)` appears in `processed` and 0 otherwise. The vocabulary list is then displayed alongside the feature vector in a `table`. To see what vocabulary words are in the email, hover the cursor over 'features', click down arrow when it appears, then enter '1' for both the largest and smallest value to select between (or sort from largest to smallest). You should see the 48 words from the dictionary that were in found in the first spam email sample.

```
features = double(ismember(vocab,processed));
table(vocab,features)
```

If desired, you can load and process a different email to see processed result and the resulting feature vector (there is no need to re-import the vocabulary list.

Train an SVM for spam classification using `fitclinear`

In this section we train an SVM classifier model for spam detection using the `fitclinear` function. As there are a large number of variables (1,899), the `fitclinear` function will offer improved performance over `fitcsvm`. (As has been mentioned in previous companion scripts, `fitrlinear` and `fitclinear` are more efficient when training models with a large number of variables). Run the code below to load the email training set in `spamTrain.mat` which contains a feature matrix X and response vector y. The `fitclinear` function is then called to train an SVM model by setting the `'Learner'` option to `'svm'`. We will also add regularization and cross-validate the model using hold-out validation (since there is a large amount of training data in this particular example) and then obtain the optimal regularization strength using hyperparameter optimization. Lastly we will load the test set and compute the training and test accuracies using the optimized classifier.

```matlab
load spamTrain.mat;
opts = struct('Holdout',0.3);
linSVMmdl = fitclinear(X,y,'Learner','svm','Regularization','ridge','OptimizeHyperparameters',{
load('spamTest.mat');
fprintf('Training Accuracy: %f | Test Accuracy: %f\n',...
        mean(predict(linSVMmdl,X)==y)*100,...
        mean(predict(linSVMmdl,Xtest)==ytest)*100);
```

List the top predictors for spam

As in ex6, we print out the top predictor words as given by their score (model coefficient). The model coefficients are found in the `Beta` model property of a `classificationLinear` variable, which we combine with the vocabulary words into a `table` before sorting. Compare with your results from ex6:

```
our click remov guarante visit basenumb dollar will price pleas nbsp
most lo ga dollarnumb
```

Figure 12: Top predictors for spam email

```matlab
% Sort the weights and obtain the vocabulary list
tbl = table(vocab,linSVMmdl.Beta,'VariableNames',{'Word','Score'})
tbl = sortrows(tbl,'Score','descend')
```

Classify your own email

The code below has been set up for you to use your trained model to classify email samples using the trained SVM model and the local function `preprocess`. Just add your own email text file to the Current Folder, add the file name (e.g. 'myemail.txt') to the code below, and select 'yourEmail' using the dropdown menu. Alternatively, you can select and classify the samples included with ex6.

```matlab
yourEmail = '';
eml = 'emailSample1.txt';
processed = preprocess(eml);
features = ismember(vocab,processed);
if predict(linSVMmdl,double(features)')
    disp('This email is probably spam.')
else
    disp('This email is probably not spam.')
end
```

Local Functions

decisionBoundary

The `decisionBoundary` function plots the positive (y = 1) and negative examples of a given dataset provided as an input feature matrix X and response vector y. The decision boundary for the classifier model `mdl` is also plotted using the `contour` function. Any misclassified points are plotted as red x's.

```matlab
function decisionBoundary(mdl,X,y)
```

```matlab
    figure; hold on;
    x1 = linspace(min(X(:,1)),max(X(:,1)));
    x2 = linspace(min(X(:,2)),max(X(:,2)));
    [Xgrid,Ygrid] = meshgrid(x1,x2);
    Z = reshape(predict(mdl,[Xgrid(:),Ygrid(:)]), size(Xgrid));
    contour(Xgrid,Ygrid,Z,'Levels',1);
    plot(X(y==1,1),X(y==1,2),'k+','MarkerSize',7);
    plot(X(~y,1),X(~y,2),'ko','MarkerFaceColor','y','MarkerSize',7);
    misclassidx = y~=predict(mdl,X);
    plot(X(misclassidx,1),X(misclassidx,2),'rx')
    hold off;
end
```

preprocess

`preprocess` is executes the steps for preprocessing emails used in the Live Script above.

```matlab
function contents = preprocess(file)
    contents = string(fileread(file));
    contents  = lower(contents);
    contents  = eraseBetween(contents,'<','>','Boundaries','inclusive');
    contents = replace(contents,"$","dollar ");
    contents = split(contents);
    contents(startsWith(contents,["http://","https://"])) =  "httpaddr";
    contents = regexprep(contents,'.+@.+',"emailaddr");
    contents = regexprep(contents,'[0-9]+',"number");
    contents = regexprep(contents,'[^a-zA-Z0-9]',"");
    contents(strlength(contents)<=1) = [];
    contents = arrayfun(@(str)string(porterStemmer(char(str))),contents);
end
```